1. Core Spring Boot Annotations
@SpringBootApplication
Combination of:

- @Configuration
- @EnableAutoConfiguration
- @ComponentScan
  Use: Main class of your Spring Boot app.

@Async

Here is the **clearest, complete, senior-level explanation** of **how @Async works in Spring Boot** — including:
✓ What it really does
✓ How Spring creates async proxies
✓ How thread pools are involved
✓ How methods run on separate threads
✓ Why @Async does **not** work in some cases
✓ Internal flow + examples
You will understand everything end-to-end.

⭐ **1. What is @Async?**
@Async tells Spring:
"Run this method on a **different thread** from a thread pool.
Don't block the caller."
This is Spring's built-in asynchronous execution mechanism.

⭐ **2. Key Idea Behind @Async**
Spring **creates a proxy** around your bean.
When you call a method annotated with @Async:
👉 **The proxy:**

- intercepts the call
- submits the method to a **thread pool**
- returns immediately
- actual logic executes in background

So your code becomes **non-blocking**.

⭐ **3. Minimum Setup for @Async to work**
You **must** enable it:
@EnableAsync
@SpringBootApplication
public class DemoApplication {}
Then annotate any method:
@Async

```
public void sendEmail() {
   // runs in background thread!
}
```

⭐ **4. How @Async works internally (step-by-step)**

✓ **Step 1 — Spring scans beans**

When Spring sees @EnableAsync, it installs an **AsyncAnnotationBeanPostProcessor**.
This processor scans all beans for methods annotated with:

- @Async

✓ **Step 2 — Spring creates a proxy of your bean**

Instead of injecting your real object, Spring injects a **proxy object**.
This proxy wraps your method call.

✓ **Step 3 — When @Async method is called**

Example:
emailService.sendEmail();
You think this calls your method directly.
But actually:

➞ It calls the **proxy**

➞ Proxy captures the method call

➞ Proxy submits it to an Executor (thread pool)

✓ **Step 4 — Method runs in Background Thread**

Spring uses a thread from:
SimpleAsyncTaskExecutor (default)
or
TaskExecutor you provide
Method runs on a worker thread, not main thread.

✓ **Step 5 — Caller does NOT wait**

emailService.sendEmail(); // returns immediately
sendEmail() starts running in parallel.

⭐ **5. Default Executor used by @Async**

If you don't configure anything, Spring uses:
SimpleAsyncTaskExecutor

⚠ **This is NOT a real thread pool.**

It creates a **new thread every time** → risky.
So you should define your own pool:

```
@Configuration
@EnableAsync
public class AsyncConfig {
```

```
  @Bean(name = "taskExecutor")
  public Executor taskExecutor() {
    ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
    executor.setCorePoolSize(10);
    executor.setMaxPoolSize(50);
    executor.setQueueCapacity(100);
    executor.initialize();
    return executor;
  }
}
```

Then use:

```
@Async("taskExecutor")
public void sendEmail() { ... }
```

⭐ 6. Return types of @Async

✓ 1. void

Simply runs async and returns immediately.

✓ 2. Future<T> or CompletableFuture<T>

Allows returning results asynchronously:

```
@Async
public CompletableFuture<String> process() {
  return CompletableFuture.completedFuture("done");
}
```

⭐ 7. Where @Async does NOT work (VERY IMPORTANT)

❌ Case 1 — Calling @Async method from the same class

Example:

```
public class EmailService {

  @Async
  public void sendMail(){}

  public void trigger() {
    sendMail();  // ❌ runs synchronously!
  }
}
```

Why?

Because proxy is bypassed when calling internally.

❌ Case 2 — @Async on private methods

Proxies can only intercept **public** methods.

❌ Case 3 — Missing @EnableAsync

It won't work at all.

❌ **Case 4 — Final class or final method**
Spring can't proxy final methods.

⭐ **8. Real Example (Easy to Understand)**
Controller:

```
@GetMapping("/start")
public String start() {
   service.doJob();
   return "Started!";
}
```

Service:

```
@Async
public void doJob() {
   System.out.println("Thread: " + Thread.currentThread().getName());
   Thread.sleep(5000);
   System.out.println("Done!");
}
```

Output:

```
Thread: taskExecutor-1
(5 sec later)
Done!
```

Controller returns immediately.
Job runs in background.

⭐ **9. Why use @Async?**
✓ **To not block HTTP request**
Send email
Send SMS
Update audit logs
Upload file
Generate reports
Send Kafka events
Call slow APIs
✓ **To parallelize tasks**
Run multiple tasks in parallel using same bean.

⭐ **10. How @Async works in restricted systems / enterprise**
In secure enterprise systems (like Daimler, BMW, banks, insurance):
@Async is used for:

- long background validations
- asynchronous logging
- auditing
- event-driven tasks

- notifications
- post-processing
- slow backend calls
- heavy DB tasks

It prevents slowing down main flow.

⭐ **11. Final Summary (Interview Ready)**

@Async tells Spring to run a method in a separate thread taken from a thread pool. Spring achieves this by creating a proxy around your bean and submitting the method invocation to an Executor.
The caller returns immediately while the method executes asynchronously.
@Async requires @EnableAsync, works only on public methods through proxy, and needs proper thread pool configuration for production.

If you want, I can also give you:
✓ Difference: @Async vs CompletableFuture
✓ Difference: @Async vs Virtual Threads
✓ How @Async works with WebClient
✓ Real Daimler/Mercedes-style example
Just tell me!

@Transactional

Here is the **clear, deep, interview-ready** explanation of **@Transactional** in Spring — what it is, how it works internally, real examples, common mistakes, propagation levels, isolation, rollback rules, proxy mechanism, and how databases actually commit/rollback.
I will explain it in a way that is easy to understand *and* perfect for senior interviews.

⭐ **1. What is @Transactional?**

@Transactional is a Spring annotation that tells Spring:
"Start a database transaction before this method runs.
If everything is successful → COMMIT.
If something fails → ROLLBACK."
It guarantees:

- ACID properties
- Consistency
- Automatic rollback
- No partial updates

⭐ **2. Why do we need a Transaction?**

Suppose you transfer money:

1. Debit account A
2. Credit account B

If debit succeeds but credit fails → database must rollback debit.
Without @Transactional, partial work happens → data corruption.
Transactions ensure atomicity.

⭐ **3. What @Transactional actually does (simple)**

When a method is annotated with @Transactional:

1. Spring **creates a proxy** of your class
2. When the method is called → proxy intercepts it
3. Proxy asks the **TransactionManager** to:
    - Open a transaction
4. Your method runs inside DB transaction
5. If method completes normally → commit()
6. If an exception occurs → rollback()

⭐ **4. Internal Flow (very important)**

Let's say:

```
@Transactional
public void placeOrder() {
  saveOrder();
  takePayment();
  updateInventory();
}
```

Step-by-step internal flow:

✓ Step 1: Call goes to Spring proxy
✓ Step 2: Proxy calls transactionManager.begin()
✓ Step 3: Method executes
✓ Step 4: If it finishes normally → commit
✓ Step 5: If exception → rollback
✓ Step 6: Proxy closes transaction

⭐ **5. Important: Only PUBLIC methods work**

Because Spring creates **proxies**, which can only intercept **public** method calls.

❌ **@Transactional does NOT work on:**

- private methods
- protected methods
- package-private methods
- methods called inside the same class ("self-invocation")

Example:

```
public class Service {

  @Transactional
  public void a() { b(); }

  @Transactional
```

```
    public void b() { ... }
}
```
Calling b() from inside a() → NOT transactional because proxy is bypassed.

⭐ 6. What triggers a rollback?

By default:

- Spring rolls back **only on runtime exceptions** (RuntimeException, Error)

Does NOT rollback on:

- checked exceptions
- Exception
- IOException
- SQLException

But you can change it:
@Transactional(rollbackFor = Exception.class)

⭐ 7. Transaction Propagation (Super Important!)

Propagation defines **how transactions behave when method A calls method B**.

1️⃣ REQUIRED (default)

If transaction exists → join it
If not → create one
Most commonly used.

2️⃣ REQUIRES_NEW

Always create a **new transaction**
Suspend existing one
Example use case:

- Logging
- Audit
- Notification
  Even if main transaction fails → logs still commit.

3️⃣ MANDATORY

Must run inside existing transaction
If no transaction → error

4️⃣ SUPPORTS

If transaction exists → join
If not → run without transaction

5️⃣ NOT_SUPPORTED

Do NOT run in a transaction
Suspend existing one

6️⃣ NEVER

Fail if any transaction is active.

7️⃣ **NESTED**
Creates nested transaction (savepoint)
Rollback only inner part

⭐ **8. Transaction Isolation Levels**
Controls how concurrent transactions see each other's changes.
Isolation Types:

Level	Prevents
READ_UNCOMMITTED	Nothing
READ_COMMITTED	Dirty reads
REPEATABLE_READ	Non-repeatable reads
SERIALIZABLE	Phantom reads

Default (in most DBs):
READ_COMMITTED

⭐ **9. Real Example in Spring Boot**

```
@Transactional
public void createOrder() {
   orderRepository.save(order);
   paymentClient.charge();
   inventoryService.reduceStock();
}
```

If any step fails → entire operation rolls back.

⭐ **10. Common mistakes (very important)**
❌ **1. @Transactional on private method**
Doesn't work.
❌ **2. Calling transactional method from same class**
Proxy is bypassed → transaction ignored.
❌ **3. Using @Transactional in controllers**
Bad practice.
Use in **service layer**.
❌ **4. Forgetting to handle checked exceptions**
Spring does NOT rollback on checked exceptions unless configured.

⭐ **11. What transaction manager does Spring use?**
Based on your database:

DB/ORM	TransactionManager
JPA / Hibernate	JpaTransactionManager
JDBC	DataSourceTransactionManager

DB/ORM	TransactionManager
Mongo	MongoTransactionManager

⭐ **12. How transaction actually works inside DB?**

Database creates:

- a log
- a lock
- a rollback segment

When commit:

- DB flushes log
- releases locks
- marks data durable

When rollback:

- DB restores from rollback segment
- releases locks

Spring only instructs the database — DB does the real transaction.

⭐ **13. Final Interview-Ready Summary**

@Transactional tells Spring to start a database transaction when a method begins, and commit or rollback depending on the outcome.
It works using proxies, which intercept the call and coordinate with a TransactionManager.
Only public methods via external calls are transactional.
Default propagation is REQUIRED and default rollback happens for runtime exceptions.
Transactions ensure atomicity, consistency, and prevent partial database updates.

If you want, I can also explain:
✓ Transaction propagation with examples
✓ Why transactional annotation sometimes doesn't work
✓ How Spring manages transactions with multiple microservices
✓ @Transactional with Kafka or messaging systems
Just tell me!

🔵

2. Bean & Dependency Injection

@Component
Generic Spring bean.
@Service
Business logic class.
@Repository
DAO layer, also converts DB exceptions → Spring exceptions.
@Controller
Handles web (MVC) requests, returns view (HTML/JSP).
@RestController

@Controller + @ResponseBody
Returns JSON directly → used in REST APIs.
@Autowired
Injects bean.
❗ Avoid field injection → use constructor injection instead.
@Qualifier
Used when multiple beans of same type exist.
@Bean
Manually create a bean in @Configuration class.
@Configuration
Class containing bean definitions.

🔵
3. Request Mapping (REST APIs)
@RequestMapping
Generic mapping at class or method level.
@GetMapping
Handles GET requests.
@PostMapping
Handles POST requests.
@PutMapping
Handles PUT requests.
@DeleteMapping
Handles DELETE requests.
@PatchMapping
Handles PATCH requests.
@RequestParam
Extract query parameter:
GET /users?id=10
@PathVariable
Extract values from URL path:
GET /users/{id}
@RequestBody
Maps JSON → Java object.

🔵
4. Validation Annotations
These come from Jakarta Validation:
@Valid
Triggers validation on request body.
@NotNull
Field cannot be null.
@NotBlank
Cannot be empty or blank string.
@Size(min, max)

Length constraints.
@Email
Validates email format.
@Min / @Max
Numeric bounds.

🔵
5. JPA & Hibernate Annotations
@Entity
Marks class as DB table row.
@Table(name="")
Custom table name.
@Id
Primary key.
@GeneratedValue
Auto-generate ID.
@Column
Control DB column properties.
@OneToOne
,
@OneToMany
,
@ManyToOne
,
@ManyToMany
Relationship mapping.
@JoinColumn
Foreign key mapping.
@Transactional
Wraps method in a DB transaction.
@Modifying
Used with update/delete queries.

🔵
6. Lombok Annotations
To reduce boilerplate.
@Getter / @Setter
Generates getters/setters.
@Data
@Getter + @Setter + @ToString + @EqualsAndHashCode + @RequiredArgsConstructor
@Builder
Creates builder pattern.
@NoArgsConstructor / @AllArgsConstructor
Generate constructors.

---

7. Spring Boot Utility Annotations
@Value
Inject values from properties:

@ConfigurationProperties
Bind entire property group to an object.

---

8. Actuator
@Endpoint
Custom actuator endpoint.

---

9. Async & Scheduling
@EnableAsync
Allows asynchronous execution.
@Async
Runs method in separate thread.
@EnableScheduling
Enables scheduled jobs.
@Scheduled
Run a job repeatedly:
@Scheduled(fixedRate = 5000)

10. Spring Security
@EnableWebSecurity
Enable Spring Security config.
@PreAuthorize
,
@PostAuthorize
Method-level role checks.
@Secured
Role-based security.